Toward Flexible Control of the Temporal Mapping from Concurrent Program Events to Animations

Eileen Kraemer
John T. Stasko

Technical Report 94-10
Graphics, Visualization, and Usability Center
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
(404) 853-9386
FAX: (404) 853-0673
E–mail: {eileen,stasko}@cc.gatech.edu

Abstract

As parallel and distributed computers become more widely available and used, the already important process of understanding and debugging concurrent programs will take on even greater importance. We believe that visualization can help in the process. In this paper we discuss heretofore unaddressed issues in the visualization of concurrent programs, and present the Animation Choreographer. The Animation Choreographer allows users to view, manipulate, and explore the set of alternate feasible orderings of the program execution under study, both through the Choreographer interface and in the context of the selected visualizations, thus providing the user with a variety of temporal perspectives on the computation.

1 Introduction

As parallel and distributed computers become more widely available and used, the already important process of understanding and debugging concurrent programs will take on even greater importance. One hope for helping this process is the use of visualization and animation tools.

A number of systems providing visualizations of concurrent programs have been developed[KS93]. Several authors[Sto89], [LMCF90], and [CHK92], have emphasized the value of displaying alternate orderings of a program's execution. However, we believe that a number of critical, interrelated issues have yet to be addressed in depth. Below we describe a few scenarios that are symptomatic of these issues. Particular systems may address one or two of these issues, but a comprehensive framework covering all has yet to be developed. This paper describes our work in developing such a framework.

- Suppose that we receive timestamped events from a parallel program and that each event activates a corresponding animation action. If two events differ by a very small time offset, should their animations be depicted sequentially, concurrently, or with partial overlap? There are probably good reasons to be able to display the animations using any of the three possibilities. For example, the activities may logically be thought of as concurrent, so we want to animate them together. Or, we may want to view the animation with respect to global execution times, thus separating the two animation actions. What if the animation responses to the different actions have different durations? How should this be presented?

- Suppose that for the two events above, the earlier one has a corresponding animation of 30 frames in duration and the latter has a 2 frame animation. How does this affect the resultant composite animation shown?

- Now suppose that our animation of these events consists of multiple view windows. In a second, different view than the one above, the first event's animation takes 5 frames and the second event's animation takes 20 frames. How are synchronizations across views coordinated?

- Suppose that we are developing an animation to help with performance evaluation and optimization. Do we make the animation properly reflect relative timings within the program? If we do so, our animation will probably include both very lengthy delays when nothing happens, a possible "waste of time" for the programmer using the animation as a debugging aid, and short bursts of high activity too rapid for the viewer to comprehend. How can we achieve the best of both worlds? That is, how can we properly reflect relative program times and also remove uninteresting sequences?

- Suppose that an animation we develop maps time to a geometric dimension, such as often seen in history chart displays. Does this geometric dimension reflect program times or animation times? If it reflects relative animation times,

the presence or absence of other animation views as described above may radically affect the history view presented.

These issues together with many others are symptomatic of the problem of specifying the temporal mapping from program actions to their corresponding animation actions. In software visualization and animation of *serial* programs, this mapping is much simpler. A program event occurs, then we initiate, run, and terminate its corresponding animation. The mapping is a simple one-to-one correspondence between two sequential streams. (Of course, we still may wish to scale animation times and durations to properly reflect program times, but that process is relatively straightforward.)

In developing an animation of a concurrent program, we may wish to have the animation reflect global times, logical times, some serialization of the events, or even some intermediate combination of these possibilities. Essentially, we want to be able to see any valid, *feasible* execution of the program, and we want its animation to match our mental model of the execution. Of course, the generation and display of *every* feasible execution of the program would be unmanageable. Instead, it would be beneficial to provide several useful, canonical orderings, based on the synchronization events produced by the program execution under study. An additional feature would be to allow the user to tweak these to produce any additional orderings that are desired.

The ability to manipulate the order of display events can be useful in a number of situations. Appropriate reorderings can produce more comprehensible displays that can allow the user to detect anomalous events more easily. In addition, providing the user with a number of displays representing several alternate feasible event orderings can provide additional perspectives on the computation. These various temporal perspectives can present unanticipated, and perhaps problematic, event sequences, and lead to further investigation of both the correctness and efficiency of the program. Finally, it may be desirable at times to ignore the actual dependences and instead, allow the user to specify the relationships that determine the order in which events are to be displayed.

Unfortunately, current systems do not provide this type of total control over the mapping, nor do they integrate it with a flexible, powerful animation system. Our goal in this research has been to understand the issues involved in the mapping and develop models that capture desired viewing behaviors. We also have sought to provide simple, easy-to-use end-user customization and flexibility of the mapping, thus generating animations that are illuminating and informative. With such capabilities, we believe that visualization and animation systems can become integral parts of concurrent software development environments. The manifestation of our work is a tool called the *Animation Choreographer* which plays a key role in the PARADE concurrent programming animation environment we are developing.

2 An Illustrative Example

Our goal is to visualize the execution of programs. This section explores a brief example to help illustrate issues in the temporal mapping from program events to animation events. We begin by introducing some terminology that will be used throughout the article. A *program event*, also known as an "interesting event", is a point in the program's execution at which information is recorded. This recorded information is referred to as an *event record*. The user specifies a mapping from each program event to zero or more *display events*. A display event is composed of changes to the graphical object in the display - objects may appear, disappear, move, grow, change color, etc. These displays are created using POLKA[SK93], an animation system that supports concurrent, overlapping animation actions that properly reflect the concurrent operations occurring in a program.

We wish to animate these displays in a consistent, comprehensible manner, and we are particularly interested in the representation of time, duration, and event order. As an example, we introduce a parallel version of an mst (minimum spanning tree) algorithm executing on the KSR-1 machine. The input to the program is a list of vertices and a list of all edges between the vertices. The output is a list of the edges required to connect these vertices into a tree such that the sum of the lengths of the edges is minimized. In this program, there is a master thread, and many slave threads. The vertices of the graph are divided up among the slave threads. The algorithm proceeds in rounds in which the slave threads nominate their shortest edge to the spanning tree, the master thread selects the shortest edge from among the nominees, then it adds the edge and corresponding vertex to the tree. The program has been instrumented to produce event records similar to those shown below:

```
A partial event trace for the master thread:

INIT 0 3500             /* an init event       */

BARRIER_IN 0 3706      /* let slaves begin    */

BARRIER_OUT 0 3706     /* wait for slaves     */
ADDTREEVERTEX 0 4186 0 /* select first vertex */
BARRIER_IN 0 4186      /* let slaves nominate */

BARRIER_OUT 0 4186     /* wait for nominations*/
...

A partial event trace for slave thread 1:

BARRIER_IN 1 3767      /* wait for the master */
NOMINATE 1 4186 3 0    /* nominate best edge  */
BARRIER_OUT 1 4186     /* let master select   */

BARRIER_IN 1 4186      /* next round begins...*/
...
```

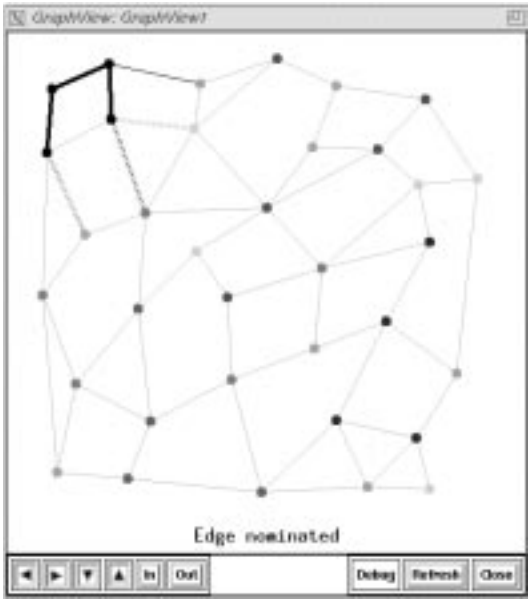

Figure 1: A minimum spanning tree visualization - graph view.

```
A partial event trace for slave thread 2:

BARRIER_IN 2 4106      /* wait for the master */
NOMINATE 2 4186 29 0   /* nominate local best */
BARRIER_OUT 2 4186     /* let master select   */

BARRIER_IN 2 4186      /* next round begins...*/
BARRIER_OUT 2 4186     /* let master select   */
...
```

Most event records that we will deal with are of this general format: an event type, a thread or processor id, an optional timestamp, and some parameters. For example, in the event record `NOMINATE 2 5186 29 0`, `NOMINATE` is the event type, `2` is the thread id, `5186` is the timestamp, and `29` and `0` are parameters describing the edge nominated. Some of these event records, such as BARRIER_IN and BARRIER_OUT, are distinguished as *synchronization events*. The presence of synchronization events permits the calculation of a partial order based on the dependences between events. At a BARRIER_IN, the slave threads block until the master thread executes a BARRIER_IN. At a BARRIER_OUT, the master thread blocks until every slave has executed a BARRIER_OUT. Thus, there is a dependence from every BARRIER_IN event executed by the master thread to the corresponding BARRIER_IN on each slave thread. Similarly, there is a dependence from every BARRIER_OUT event executed by each slave thread to the corresponding BARRIER_IN event executed by the master. These dependences, plus the serial dependence between successive events executed by each individual thread, are used to construct a partial order.

Figure 1 represents a view of the execution of this program, created using POLKA. Initially, each edge is thin and gray, and each node is colored to indicate the thread responsible for it. When a thread nominates its shortest edge to the current vertex,

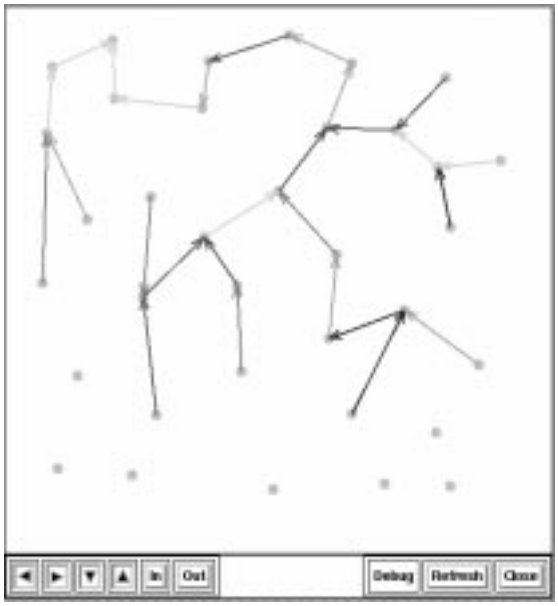

Figure 2: A minimum spanning tree visualization - graph attribute view.

the edge thickens, and flashes in that thread's color. When the master selects the best, the selected edge becomes thick and black, and all other nominated edges return to their thin, gray state. The thick black tree can be seen forming as the visualization progresses.

Figure 2 represents a different view of the program's execution. This view presents a data structure called *closest* maintained by the algorithm. In essence, *closest* keeps the best (shortest) edge from each vertex into the minimum spanning tree. That is, at any given time, a particular vertex may connect into the spanning tree via a number of different edges. Only one of those edges will be shortest in length, and that edge will be the best way to reach the spanning tree. em Closest keeps track of these best edges. This view is useful for identifying problems related to changes in this important data structure.

There are several orderings under which the user may wish to view these displays. In the following paragraphs we present some terminology that will assist us in discussing these orderings. We then show the appearance of the Choreographer, and discuss the appearance of the mst displays under each orderings. Finally, we discuss the benefits and consequences of each selection.

We use the term *execution time* to refer to physical timestamps, and the term *animation time* to refer to animation frame number. A *logical time* ordering is any event ordering that does not violate the dependences imposed by the serial execution within a thread (or process) and the synchronization events across threads. The default orderings that we provide are *timestamp*, *adjusted timestamp*, *serialize*, and *maximum concurrency*.

An initial ordering choice might be a *timestamp* ordering - the execution times are used to order the events for visualization. Figure 3 shows the appearance of the choreographer using the sample events presented earlier, and a timestamp ordering.

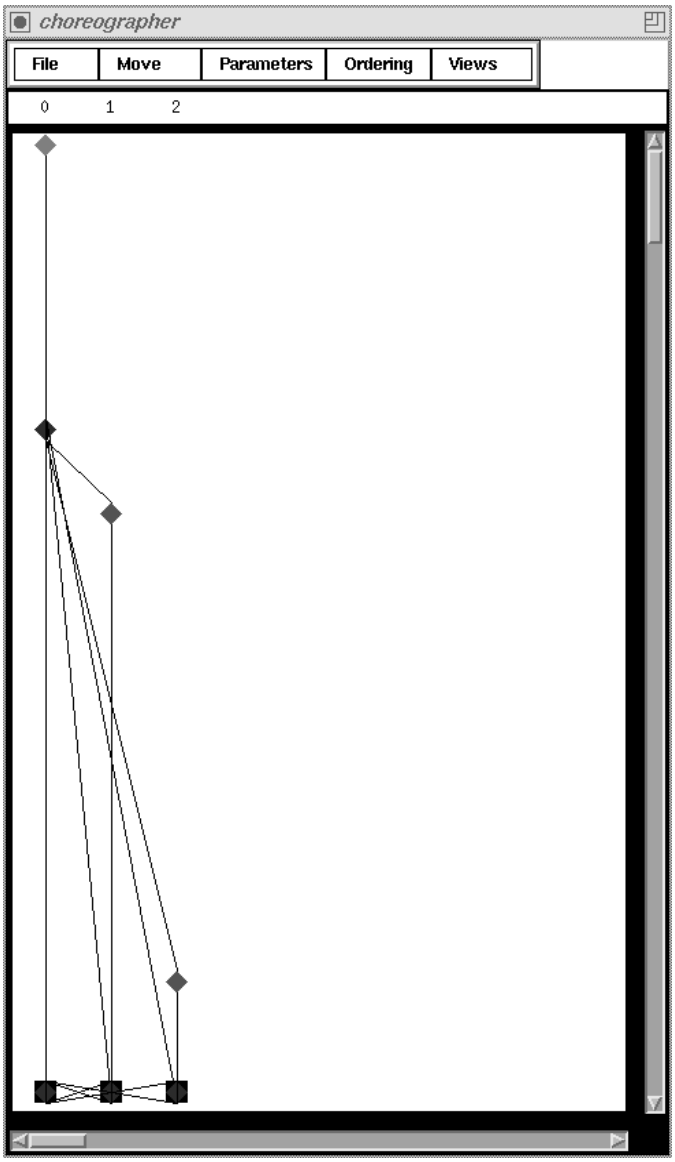

Figure 3: The choreographer display, under timestamp ordering.

This ordering is frequently very useful to a user wishing to see the actual order of execution, when such information is available. This method relies on the existence of a global clock with adequate resolution, and will produce an essentially sequential visualization under these circumstances. Poor resolution, or timestamps that are not valid across processors, however, may produce visualizations that are misleading or incorrect.

An examination of the mst event traces above reveals duplicate timestamps - the clock used was not of adequate resolution. The overlapping event symbols in the Choreographer display are a result of duplicate timestamps. In a timestamp ordered visualization, all events with the same timestamp are animated concurrently. In the mst display under timestamp ordering, it may appear that an edge has been simultaneously nominated and selected - a misleading representation of the program's execution.

Within timestamp ordering we have several choices for scaling. We can use a 1:1 mapping from timestamp units to animation frames. However, this can result in an animation with long periods of inactivity, punctuated by short bursts of activity too rapid for the viewer to comprehend. Another option is to use an n:1 mapping from timestamp units to animation frames. This shortens the spans in which nothing happens, but intensifies the short bursts of activity. A third option is to use the timestamps to order the events for visualization, but to ignore them in determining the interevent waiting time. This eliminates the long waits, and allows the event activity to be visualized at a rate the viewer can understand. However, in this type of scaling we lose information about the relative timing of the program events. Ideally, it would be desirable for the mapping from execution time to animation time to behave like a "fun-house mirror." That is, we would like to compress long inter-event times, and stretch out periods of high activity, allowing viewers to discriminate between individual events, but preserving a perspective on the actual timing of events.

The choreographer display under an *adjusted timestamp ordering*, shown in Figure 4, represents a compromise on these goals. In this ordering the timestamps are adjusted just enough so that causal ordering is maintained, but long interevent times are unaffected. This ordering is useful in obtaining a valid visualization without losing the perspective on the true sporadic nature of the program's execution behavior.

Figure 5 shows the choreographer under a *serial ordering*. For a serial ordering, we construct a complete ordering of events consistent with the partial order determined by the dependence relations. This method can produce valid, comprehensible visualizations in the absence of globally synchronized timestamps with adequate resolution, such as we have in this example. In the mst display, a serial ordering will produce a visualization in which all edge nominations in a round are animated before the edge selection is animated. In this sense, the visualization is "correct." We lose the long interevent times.

Finally, we may wish to use a *maximum concurrency* ordering, as illustrated by the Choreographer display of Figure 6. In this ordering, we gather all events that *could* have occurred together, and animate them simultaneously. Essentially, this view shows the maximum concurrency possible given the partial order defined by the synchronization events. In the mst display this ordering produces a visualization that

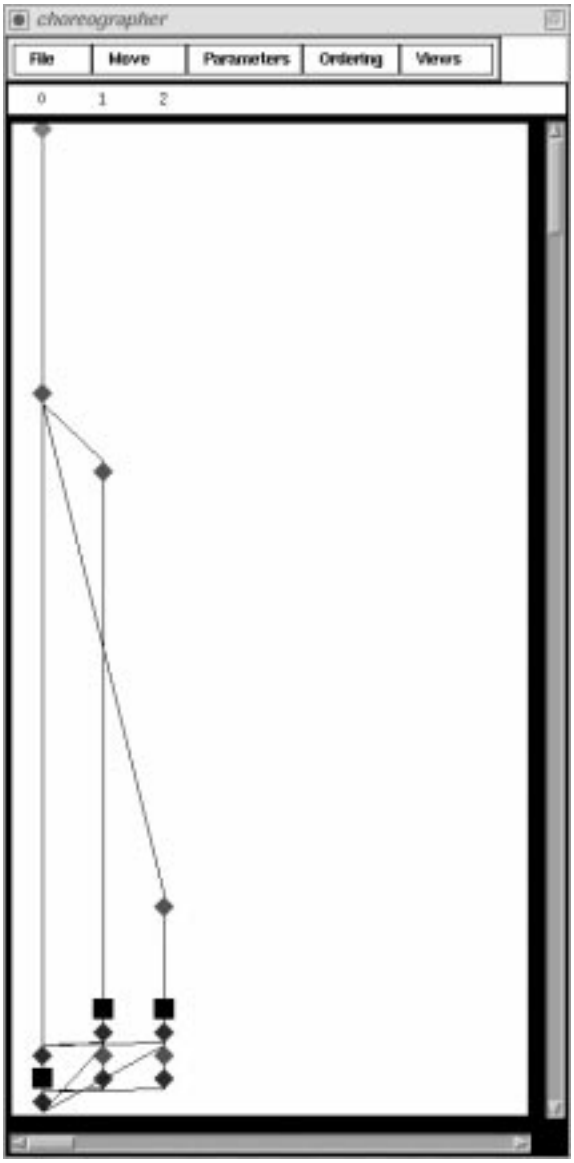

Figure 4: The choreographer display, under adjusted timestamp ordering.

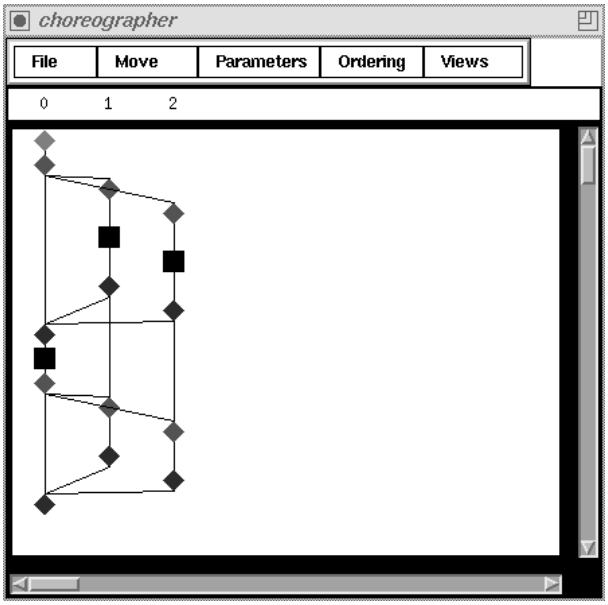

Figure 5: The choreographer display, under serialized ordering.

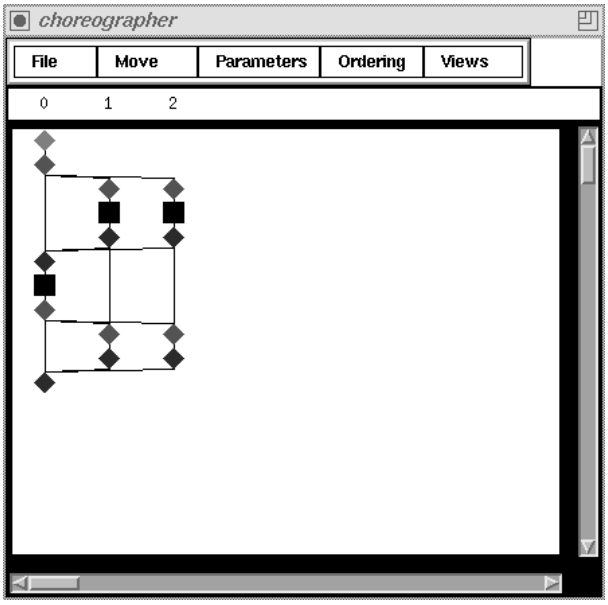

Figure 6: The choreographer display, under maximum concurrency ordering.

matches our mental model of the computation. We see the algorithm proceeding in rounds - all edges nominated in the same round are shown flashing at the same time, one is selected and becomes thick and black, all those not selected simultaneously return to their thin, gray state, and then another round begins. In other visualizations we have found this ordering type to be useful for identifying bugs by illuminating concurrent situations that were not imagined by the program's designer.

These various temporal perspectives can provide the user with insight into the program's execution, with each different ordering of the animation shedding light on a different aspect of the computation. The Animation Choreographer allows users to view program animations under the orderings described above, and to specify variations on these orderings. In the following sections we describe how the Choreographer tool functions and what a programmer must do to use it.

3 PARADE and the Animation Choreographer

The Animation Choreographer is a component of the PARADE (PARallel Animation Development Environment) system for the visualization of concurrent programs. PARADE contains three major components - some form of program instrumentation or monitoring, the POLKA animation system, and the Animation Choreographer.

The use of an instrumentation or monitoring tool, which will vary between architectures and languages, helps identify the event records for a program. Currently under development is an automatic instrumentation tool for parallel FORTRAN, and monitoring or tracing libraries for KSR Pthreads and Conch, a PVM-like distributed system. The use of such a tool is not required, however. PARADE accepts events records in various formats.

The POLKA[SK93] animation toolkit, which has both 2-D and 3-D versions, is used to design and generate visualizations. POLKA supports *true* animation - smooth, continuous movements and actions, not just blinking objects or color changes. It supports concurrent, overlapping animation actions on multiple objects. Thus, it can properly reflect the concurrent operations occurring in a parallel program. POLKA is available via anonymous ftp from `par.cc.gatech.edu`.

Using POLKA, libraries of visualizations have been developed - synchronization, history, and and callgraph views for Pthreads programs on the KSR, performance views based on PICL[GHPW90] traces, 3-D visualizations of communication on the MasPar, algorithmic and performance views of branch and bound algorithms in the iPSC hypercube, as well as a number of application-specific visualizations. Using PARADE, programmers may select visualizations from libraries such as these, or they may create their own new visualizations.

The functions performed by the Animation Choreographer rely heavily on the types of event records that it will process and the displays that it will control. Thus, a view-specific, trace-format specific version of the Choreographer is generated at the beginning of a visualization session. This generation is performed by the Choreographer-generator tool. TraceView[MHJ91] also uses this concept of a visualization session. Figure 3 illustrates the generation process in PARADE. The user

supplies three files an *event spec* file, an *anim spec* file, and a *map spec* file. The generator tool then produces source code representing the session-specific portions of the Choreographer. This source code is then compiled and linked with the object code of the generic portions of the Choreographer*, and with the object code of the animations, which can be either programmer-created or taken from the libraries of default views. Below, we describe the contents of the specification files.

The event spec file defines the format of the event records that the program produces. A sample event spec file, corresponding to the program event records introduced earlier in the paper, is shown below. The first two lines specify the set of parallel programming constructs in use and the position number of the event type field in each record. In this example, KSR_C indicates the set of constructs available in the pthreads package on the KSR-1 machine. This information is needed to correctly interpret the ordering semantics of the synchronization events.

```
KSR_C
1
INIT:id           pid:d           ti:d  title:s
VERTEX:id         pid:d           ti:d  vnum:d       xpos:f    ypos:f
EDGE:id           pid:d           ti:d  fromvert:d   tovert:d
ADDTREEVERTEX:id  pid:d           ti:d  vertnum:d
ADDTREEEDGE:id    pid:d           ti:d  fromvert:d   tovert:d
NEXTPHASE:id      pid:d           ti:d
RESPONSIBLE:id    pid:d           ti:d  vertnum:d
CLOSEST:id        pid:d           ti:d  vertnum:d    closest:d
NOMINATE:id       pid:d           ti:d  fromvert:d   tovert:d
BARRIER_IN:id     BARRIER:_synch pid:d ti:d
BARRIER_OUT:id    pid:d           ti:d
```

The remaining lines show the event record format for each of the event types produced by this program. Each field specification takes the form *label:type*. Some labels and type names have special meaning. For example, the *id* type in *INIT:id* indicates that this is the event type field, and that the string *INIT* will be found in the event record. The labels *pid* and *ti* are also reserved, and indicate the thread id field and timestamp field. The type specification here refers to the data type: *d* indicates integer, *f* is float, *c* is character, *s* is string, and so on. Additional parameters have user-assigned labels, such as *index* and *value* in the INPUT record definition.

The BARRIER record definitions contains the specification for a special type of event, a synchronization event. This is indicated by the *_synch* type of the second item. The label, "BARRIER" is a predefined synchronization type in our library of KSR_C type ordering primitives. Other types of synchronization events with predefined semantics are process forks and joins, mutex locks and unlocks, condition wait, signal and broadcast, and shared variable access. These have been developed for KSR pthreads programs and for a cthreads package in use at Georgia Tech. These ordering

*The generic portion of the Choreographer is approximately 1100 lines of C++, and the generator tool produces approximately 900 lines of C++ source code representing the session-specific portions of the Choreographer. The interactive user interface and graphics portions of the Choreographer are implemented using the X Window System and Motif.

Figure 7: Architecture of the PARADE system. Shaded boxes indicate files created by the end-user.

semantics are based on the "happened before" relation as discussed in [Lam78] and [Fid91].

The anim spec file defines the POLKA animation *Views* that may be created, the *scenes* associated with each view, and the parameters associated with each scene. A *View* is a POLKA class that encapsulates a particular visual representation of the program being animated. Developers subclass View to build their own program representations. Multiple views of a program (think of them simply as windows on a workstation window system) can be active simultaneously. A *scene* is is a subroutine that implements a set of changes in the display.

```
View GraphView mst.H
Init            title:s
Vertex          ti:d pid:d vnum:d xpos:lf ypos:lf
Edge            ti:d pid:d from:d to:d
ClaimVertex     ti:d pid:d vnum:d
EdgeNominate    ti:d pid:d vnum:d
AddEdgeToTree   ti:d pid:d from:d to:d
AddVertexToTree ti:d pid:d vnum:d
NextPhase       ti:d

View GraphAttrView mst.H
Init            title:s
Vertex          ti:d pid:d vnum:d xpos:lf ypos:lf
Arrow           ti:d pid:d from:d to:d
```

Here, two views, GraphView and GraphAttrView are defined. Both views require include file "mst.H." Eight scenes are associated with GraphView - *Init, Vertex, Edge, ClaimVertex, EdgeNominate, AddEdgeToTree, AddVertexToTree*, and *NextPhase*. We also see that GraphAttrView has three associated scenes - *Init, Vertex,* and *Arrow*.The parameters required by each scene are listed after it. For example, we see that the AddEdgeToTree scene requires two integer parameters, labeled *from* and *to* by the user. The label *ti* indicates that this is a time parameter. The actual time value used in the animation will depend on selections made by the user at display time, and the type of time values specified in the mapping specification file.

The *mapping specification* file for this example is shown below. The map spec file defines the animation actions that are to be initiated by each program event. The first line states that the Choreographer should call the Init scene of GraphView view when it processes an INIT program event, and that parameter 3 of the program event should be used as the parameter to the scene call. A single program event can results in calls to scenes from multiple views, as illustrated by the INIT and VERTEX events. Other program events affect only one display view. For example, the EDGE affects only the Graph view, and closest affects only the GraphAttr view.

```
INIT -> GraphView.Init 3
INIT -> GraphAttrView.Init 3
VERTEX -> GraphView.Vertex ti 1 3 4 5
VERTEX -> GraphAttrView.Vertex ti 1 3 4 5
EDGE -> GraphView.Edge ti 1 3 4
ADDTREEVERTEX -> GraphView.AddVertexToTree ti 1 3
```

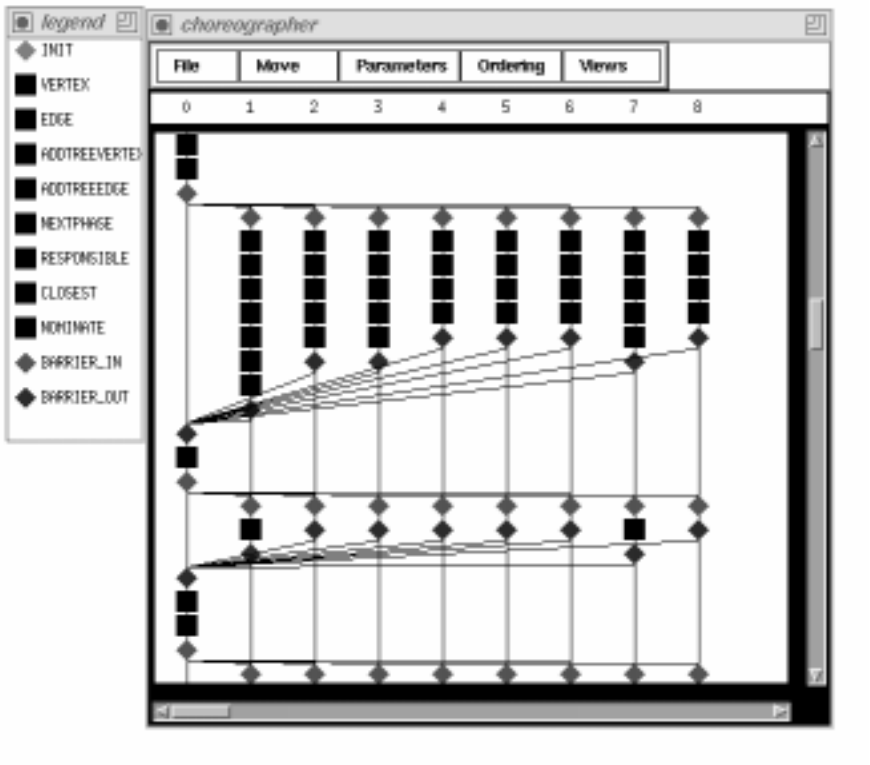

Figure 8: The user interface for the interactive run-time component of the Animation Choreographer. Each column represents a process or thread, and time moves top-to-bottom. Different shapes and colors depict different types of program events. Lines indicate ordering dependences. This interface is currently under revision; missing above are the arrowheads on the dependences arcs.

```
ADDTREEEDGE -> GraphView.AddEdgeToTree ti 1 3 4
NEXTPHASE -> GraphView.NextPhase ti
RESPONSIBLE -> GraphView.ClaimVertex ti 1 3
CLOSEST -> GraphAttrView.Arrow ti 1 3 4
NOMINATE -> GraphView.EdgeNominate ti 1 3 4
```

The parameter *ti* has special meaning, and indicates that a time value should be sent as a parameter to the animation scenes. The actual value sent will depend on the ordering and duration options chosen by the user of the Choreographer. If a timestamp ordering is in effect, the recorded timestamp will be passed on to the animation scene. Other orderings will result in a time value representing elapsed time in animation frames.

A Choreographer-generator tool reads these specification files and produces a version of the Choreographer that is specific to the types of event records and animations that have been defined. The graphical interface to the Choreographer appears as in Figure 3.

Figure 9: A dialog box, displaying the contents of an event record.

Interacting with the Choreographer

After the Choreographer has been started, the first task is to read in the event records. This is done through a selection in the File menu. An *execution graph*, similar to the graphs described in [ZR91], [HMW90] , and [ZSM92] then appears in the Choreographer display window. This is an acyclic, directed graph in which the nodes represent the recorded program events, and the arcs indicate the temporal precedence relations between these events. The events produced by a particular process or thread are displayed in a column. Arcs between these nodes indicate the sequential relationship between the events of a single process. Arcs between columns are the result of synchronization events such as forks, joins, or barrier synchronizations.

Processors (or threads, etc.) are arranged from left to right. Vertical position in the graph represents execution time, with earlier times appearing above later times. Shape and color of node objects can be used to identify differeny event types. The execution graph reflects the program events as they were recorded. The user can examine the recorded events by scrolling through the graph, and by clicking on nodes of interest. A dialog box, as shown in Figure 5, displays the event record, complete with user-defined labels.

The user then selects an automatic ordering type from the Ordering menu. The orderings include:

- *Timestamp* - The timestamps that have been recorded with the program events are used to order the events for visualization. This method relies on the existence of a global clock with adequate resolution, and will produce an essentially sequential visualization under these circumstances. Poor resolution, or timestamps that are not valid across processors may produce visualizations that are misleading or incorrect. This view is frequently very useful to a user wishing to see the actual order of execution, when such information is available.

- *Adjusted Timestamp* - The dependence relations are first used to order the events. The timestamps that have been recorded with the program events are then incremented by the minimum amount necessary for the partial order to

hold. This view is useful to a user wishing to see both the actual order of execution, and to observe the intermittent periods of activity and inactivity.

- *Serialize* A consistent *serial* ordering, based on the synchronization events, is used to order the events for visualization. This method can produce valid, comprehensible visualizations in the absence of globally synchronized timestamps with adequate resolution.

- *Maximum_Concurrency* A consistent *concurrent* ordering is used to order the events for visualization. Here, we gather all events that *could* have occurred next, and animate them simultaneously. Timestamps are not required. Essentially, this view shows the maximum concurrency possible given the partial order defined by the recorded synchronization events. We have found this perspective to be useful for identifying bugs by illuminating concurrent situations that were not imagined by the program's designer.

Other ordering types, not yet implemented, include:

- *Phase ordering* is based on user-defined phases. For example, in a message passing program, the user may wish to view the execution in terms of rounds of communication. Or the user may wish to concurrently display all messages sent by a particular process in a particular round, despite the fact that these could not have occurred concurrently. This allows the user to impose constraints, other than the temporal constraints inherent in the recorded synchronization events, on the order of animation of program events.

The user may click on nodes and drag them to new positions, altering the visualization times. The Choreographer prevents users from dragging nodes to positions that violate the temporal precedence relations.

The user selects the POLKA animation views for display from the Views menu. The views listed in the menu correspond to those defined in the anim spec file. The animation windows and a control panel then appear on the screen. Through the contorl panel the user may pause the animation, step through it, and alter the speed. Controls on the display windows allow the user to zoom and pan. The user may then wish to display additional views, manually adjust the current ordering, select a new ordering, or examine particular trace events, in order to better understand the execution of the program. This all occurs without leaving the Choreographer tool, and requires no further compilation or execution of the program under study. Also, an ordering can be saved to a file for later retrieval, display, and manipulation.

4 Related Work

Time is a crucial dimension in the analysis of parallel programs. The importance of time in representing and understanding the behavior or parallel programs has been recognized by numerous researchers. Systems such as MAD[ZR91], the monitoring system for the Makbilan shared memory machine, and TraceViewer[HMW90] display

a program causality graph, allow the user to select a node, and will highlight those event nodes that must have occurred before, must have occurred after, or may have occurred concurrently with the selected node. However, knowledge of this ordering is not used to drive the ordering of events in an associated visualization.

Stone[Sto89] and LeBlanc, et al.[LMCF90], emphasize the value of displaying both the actual order of events in a program's execution and alternate orderings of those events. Stone's concurrency map is designed to concisely represent the collection of feasible event-orderings for a set of concurrent processes. The causality graph[ZSM92] also displays a logical view of the execution of a concurrent program. The concurrency map provides a single representation of the execution of the program. Furthermore, the alternate orderings of events are not elaborated; the viewer must study the display closely to derive them. Neither the concurrency map nor the causality graph provide support for user-defined visualizations.

LeBlanc, et al.[LMCF90] define *physical time*, *logical time*, and *phase time*, and apply these concepts to visualization. They state that, "the way in which time is presented, either physical, logical, or phase time, can significantly affect the ease of program analysis." Physical time is based on timestamps, and defines the duration of the execution and the individual operations that make up the execution. Logical time, based on the happened-before relationship[Lam78] and the causal relationships in the program, defines the observable order of events, the partial order. Phase time relies on the user's specification of the events that constitute a phase, and is a refinement of logical time. Their system, Moviola, displays an execution history graph as a space-time diagram. The layout of events in the diagram can be selected to use either logical time or physical time.

There are limitations to this approach as well. Moviola is a special-purpose tool, requiring that programs use a specialized library of synchronization calls. The program is assumed to synchronize via access to locks on shared data structures. Support for user-defined visualizations is limited to the user's option to write a Lisp program to traverse the execution history graph to collect data which may be sent to some, unspecified, visualization system. No support is provided for producing visualizations which adhere to the concurrent order specified in the execution history graph.

Perspective Views[HC91] performs "reordering of events", but for a different purpose, and on a different scale. The goal of Perspective Views is the use of abstraction to understand the flow of data and control between processors. The user defines abstract events, usually logical patterns of communication. An event-recognizer processes the event stream to detect these patterns, and produce a visualization. Reordering is performed to ensure that logically related events can be displayed as distinct visual units; that is, so that abstract events that overlap in space do not overlap in time. If a sequence of events that matches the user-specified pattern is located, a picture is generated. Otherwise, the display fails. The specified order is not used to drive an animation, nor are alternate consistent orderings generated automatically. More recent work[CHK92] by the same authors extends these techniques to the manipulation of logical time in order to produce more coherent visualizations. To produce the desired ordering of events in the visualization, the user must define the appropriate abstract event on which to base the ordering, and may also introduce

a "perspective view" by selectively ignoring some of the dependencies.

5 Conclusions and Future Work

The Choreographer provides both a graphical representation of the order of events and a method by which users may examine and manipulate this representation. It supports the reordering of visualization events, allowing the user to create displays with a variety of event orderings, or *temporal perspectives*. We believe that the ability to examine the trace events both as part of an execution graph and as individual event records, to manipulate and control the speed and ordering of these events in user-defined visualizations, and in particular to view illustrative animations of the program's execution under different orderings, can greatly assist the user in understanding and exploring the program under study.

Currently, the set of reordering routines included in the Choreographer is limited to the synchronization events described in this paper, as implemented on the KSR or in the cthreads package. In the future we plan to expand this collection of ordering routines to include other synchronization events and programming models - distributed systems, message passing systems, and user-defined synchronization events and semantics.

Additionally, the interface to the Choreographer is being refined, and we are working toward improving the usability of the system. We also are developing abstraction and filtering techniques that will be necessary to examine the massive program traces that can be generated by concurrent programs.

References

[CHK92] Janice E. Cuny, Alfred A. Hough, and Joydip Kundu. Logical time in visualizations produced by parallel programs. In *Visualization '92*, Boston, MA, October 1992.

[Fid91] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, August 1991.

[GHPW90] G.A. Geist, M.T. Heath, B.W. Peyton, and P.H. Worley. PICL: a portable instruemnted communication library, C reference manual. Technical report, Oak Ridge National Labs, Oak Ridge, TN, 1990.

[HC91] Alfred A. Hough and Janice E. Cuny. Perspective views: A technique for enhancing parallel program visualization. In *Proceedings of 1991 International Conference on Parallel Processing*, pages II 124–132, August 1991.

[HMW90] David P. Helmbold, Charlie E. McDowell, and Jian-Zhong Wang. Traceviewer: A graphical browser for trace analysis. Technical Report UCSC-

CRL-90-59, Univ. of California at Santa Cruz, Santa Cruz, CA, October 1990.

[KS93] Eileen Kraemer and John T. Stasko. The visualization of parallel systems: An overview. *Journal of Parallel and Distributed Computing*, 18(2):105–117, June 1993.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LMCF90] Thomas J. LeBlanc, John M. Mellor-Crummey, and Robert J. Fowler. Analyzing parallel program execution using multiple views. *Journal of Parallel and Distributed Computing*, 9(2):203–217, June 1990.

[MHJ91] Allen D. Malony, David H. Hammerslag, and David J. Jablownski. Traceview: A trace visualization tool. *IEEE Software*, 8(5):29–38, September 1991.

[SK93] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, June 1993.

[Sto89] Janice M. Stone. A graphical representation of concurrent processes. *SIGPLAN Notices*, 24(1):226–235, January 1989. (Proceedings of the Workshop on Parallel and Distributed Debugging, Madison, WI, May 1988).

[ZR91] Dror Zernik and Larry Rudolph. Animating work and time for debugging parallel programs - foundations and experience. *SIGPLAN Notices*, 26(12):46–56, December 1991. In Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, Santa Cruz, CA, May 1991.

[ZSM92] Dror Zernik, Marc Snir, and Dalia Malki. Using visualization tools to understand concurrency. *IEEE Software*, 9(3):87 – 92, May 1992.